EXHIBIT Z

Effective Interprocedural Resource Leak Detection

Emina Torlak Satish Chandra

IBM T.J. Watson Research Center, USA
{etorlak,satishchandra}@us.ibm.com

ABSTRACT

Garbage collection relieves programmers from the burden of explicit memory management. However, explicit management is still required for finite *system resources*, such as I/O streams, fonts, and database connections. Failure to release unneeded system resources results in *resource leaks*, which can lead to performance degradation and system crashes.

In this paper, we present a new tool, TRACKER, that performs static analysis to find resource leaks in Java programs. TRACKER is an industrial-strength tool that is usable in an interactive setting: it works on millions of lines of code in a matter of minutes and it has a low false positive rate. We describe the design, implementation and evaluation of TRACKER, focusing on the features that make the tool scalable and its output actionable by the user.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Reliability, Verification

Keywords

resource leaks, alias analysis, inter-procedural analysis

1. INTRODUCTION

While garbage collection frees the programmer from the responsibility of memory management, it does not help with the management of finite *system resources*, such as sockets or database connections. When a program written in a Java-like language acquires an instance of a finite system resource, it must release that instance by explicitly calling a dispose or close method. Letting the last handle to an unreleased resource go out of scope *leaks* the resource: the runtime system gradually depletes the finite supply of system resources, leading to performance degradation and system crashes. Ensuring that resources are always released, however, is tricky and error-prone.

```
1  public void test(File file, String enc) throws IOException {
2    PrintWriter out = null;
3    try {
4      try {
5        out = new PrintWriter(
6                  new OutputStreamWriter(
7                      new FileOutputStream(file), enc));
8      } catch (UnsupportedEncodingException ue) {
9        out = new PrintWriter(new FileWriter(file));
10     }
11     out.append('c');
12   } catch (IOException e) {
13   } finally {
14     if (out != null) {
15       out.close();
16     }
17   }
18 }
```

Figure 1: Code example adapted from `ant`.

As an example, consider the Java program in Fig. 1. The allocation of a FileOutputStream on line 7 acquires a stream, which is a system resource that needs to be released by calling close() on the stream handle. The acquired stream object then passes into the constructor of OutputStreamWriter, which remembers it in a private field. The OutputStreamWriter object, in turn, passes into the constructor of PrintWriter. In the **finally** block, the programmer calls close() on the PrintWriter object. This close() method calls close() on the "nested" OutputStreamWriter object, which in turn calls close() on the nested FileOutputStream object. By using **finally**, it would appear that the program closes the stream, even in the event of an exception.

However, a potential resource leak lurks in this code. The constructor of OutputStreamWriter might throw an exception: notice that the programmer anticipates the possibility that an UnsupportedEncodingException may occur. If it does, the assignment to the variable out on line 5 will not execute, and consequently the stream allocated on line 7 is never closed.

Resource management bugs are common in Java code for a number of reasons. First, a programmer might omit a call to close() due to confusion over the role of the garbage collector. Second, even a careful programmer can easily fail to release all resources along all possible exceptional paths, as illustrated in Fig. 1. Finally, a programmer needs to understand all relevant API contracts. In the example above, the programmer correctly reasoned that closing the PrintWriter instance closes the nested resources through cascading close() calls. But cascading close is not universal across all APIs, and a programmer could easily make incorrect assumptions.

Contributions. In this paper we describe the design, implementation and evaluation of a static analysis tool, called TRACKER, for resource leak detection in Java programs. Our contribution is in overcoming the *engineering* challenges to building a useful, scalable leak detection tool. By *useful*, we mean that the reports produced by the tool must be actionable by a user, as opposed to merely comprehensive. By *scalable*, we mean that the tool must be able to handle real-world Java applications consisting of tens of thousands of classes.

Some of the challenges that we address are common to all static analysis tools and include the following:

Scalable inter-procedural analysis. Inter-procedural reasoning about aliases is crucial to building a useful resource leak detection tool. Given the program in Fig. 1, for example, the tool needs to reason that the instance of FileOutputStream released in OutputStreamWriter's close() method is in fact the same instance that was attached to the OutputStreamWriter object when the latter was constructed. The need for such reasoning can make it difficult to handle applications with tens of thousands of classes, as programs of this size are not amenable to precise whole-program alias analysis. TRACKER side-steps the general alias analysis problem by tracking pertinent aliasing information using access paths [19] in the context of an efficient inter-procedural data-flow analysis [21]. We observe, empirically, that surprisingly sparse tracked aliasing information can accord high precision.

False positives. False positives are inevitable in any tool that guarantees soundness. However, if a bug-finding tool produces too many false positives, the average user will simply ignore its results. Building a bug-finding tool as opposed to a verification tool (whose goal is to determine whether or not the code is free from a certain class of errors) opens up the possibility—and creates an expectation—of prioritizing reports that are likelier than others to be true positives. TRACKER offers such prioritization by keeping a record of whether a bug was witnessed along some path, or whether it was assumed due to the limitations of the analysis. We show, empirically, that bugs arising from analysis limitations are much more likely to be false positives.

We also tackle issues that are specific to the problem of resource management:

Nested resources. It is not enough to track just system resources, because the corrective action that a user needs to take may be to call a close method on an object that nests a resource instance. Suppose that in the above example, the constructor of OutputStreamWriter could not throw an exception, but instead, the programmer forgot to call out.close(). Even though the leaked system resource would still be the FileOutputStream, a user may justifiably expect a tool to report that the PrintWriter referenced by out should be closed rather than the nameless instance of FileOutputStream. TRACKER offers reports that make proper remedial action more apparent.

Exceptional flows. As shown in the example in Fig. 1, as well as by others previously [24], programmers often make mistakes in using **try-catch-finally** blocks. A resource leakage tool therefore needs to pay special attention to bugs lurking there. Reporting leaks due to all possible runtime errors would overwhelm the user with uninteresting results, but we must deal with those exceptions that the programmer expects to occur. For instance, a programmer would typically ignore a leak report based on an OutOfMemoryError, because most programs are not expected to deal with such abnormal conditions. Instead, we present a *belief-based* [14] heuristic that suppresses reports due to exceptional flow unless the analysis finds plausible evidence to justify the report.

Summary of Results. We evaluated TRACKER on a suite of 5 large open-source applications, ranging in size from 6,900 to 69,000 classes. All benchmarks were analyzed in a matter of minutes, with our tool identifying resource leaks at an actionable (true positive) rate of 84 to 100 percent. The engineering techniques that we describe here are crucial for actionability of the output produced by TRACKER: our exception filtering mechanism, for example, reduces the false positive rate by roughly 2.5×. In comparison to FINDBUGS, TRACKER found 5 times more true positives. Our tool has been used internally by IBM developers, who applied it on real code and fixed most of the leaks it reported. A version of TRACKER ships in Rational Software Analyzer version 7.1 as a part of its Java static analysis suite.

Outline. The remainder of this paper is organized as follows. Section 2 presents the core resource tracking algorithm of TRACKER. Section 3 presents enhancements germane to producing useful reports for such a tool. Section 4 presents selected implementation details, especially how we construct the call graph. Section 5 presents a detailed empirical evaluation. Section 6 reviews related work, and Section 7 concludes the paper.

2. CORE ALGORITHM FOR RESOURCE TRACKING

In this section, we first give a quick overview of how the basic analysis works as applied to Fig. 1. Subsequently, we give formal details of the core intra-procedural and inter-procedural resource tracking algorithms.

2.1 Overview

The analysis takes as input a control-flow graph (CFG) and a specification of procedures that acquire and release system resources. Figure 2 shows the relevant parts of the CFG for Fig. 1, along with the CFGs of some of the called methods. We introduced temporary variables t1 and t2 when constructing the CFG (a). The example's specification declares that the constructor for FileOutputStream allocates a resource, and the corresponding close() releases it.

The goal of the analysis is to establish that the allocation of a resource on line 7 is followed by its release on all possible execution paths to exit; when this property does not hold, there is a potential leak. TRACKER symbolically tracks each resource through paths in the CFG until either a) it is released, or b) it becomes unreachable without being released, and thus leaks. Consider the program path 7-6-5-11-14-15, starting with the resource allocation on line 7. The next two lines require inter-procedural tracking due to constructor calls. The constructor on line 6 stores its argu-

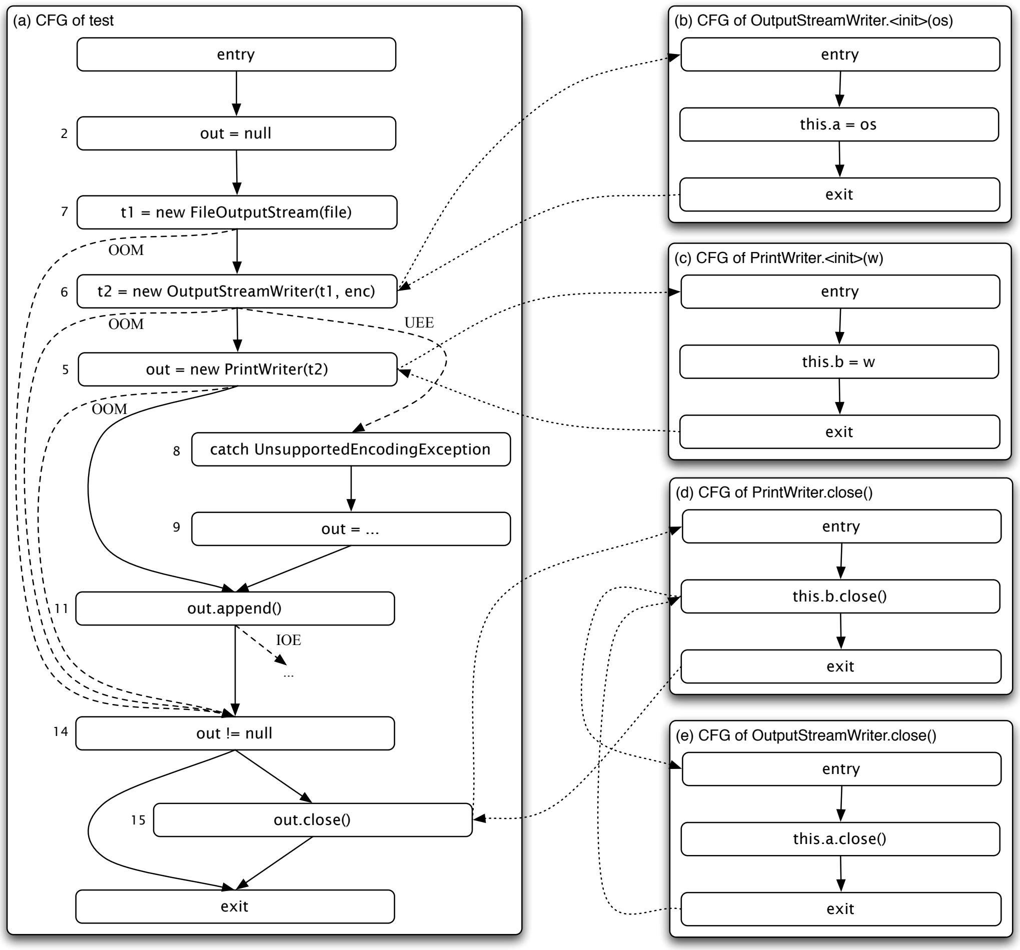
(a) CFG of test
entry
2 out = null
7 t1 = new FileOutputStream(file)
OOM
6 t2 = new OutputStreamWriter(t1, enc)
OOM
UEE
5 out = new PrintWriter(t2)
OOM
8 catch UnsupportedEncodingException
9 out = ...
11 out.append()
IOE
...
14 out != null
15 out.close()
exit
(b) CFG of OutputStreamWriter.<init>(os)
entry
this.a = os
exit
(c) CFG of PrintWriter.<init>(w)
entry
this.b = w
exit
(d) CFG of PrintWriter.close()
entry
this.b.close()
exit
(e) CFG of OutputStreamWriter.close()
entry
this.a.close()
exit

Figure 2: Control-flow graph of the procedure shown in Fig. 1. Numbers to the left are line numbers. Dashed edges represent exceptional flow labeled by the type of exception they carry, *e.g.*, OOM = OutOfMemoryError. Dotted edges represent inter-procedural control transfers.

ment into an instance field a. Our analysis concludes that after line 6, both expressions t2.a and t1 point to the tracked resource. The constructor on line 5 stores its argument into an instance field b. Our analysis, likewise, concludes that after line 5, the expression out.b.a also refers to the tracked resource.

The call out.close() on line 15 transitively calls close() on expressions out.b and out.b.a (notice that **this** in CFGs (d) and (e) would be bound appropriately), the last one releasing the tracked resource as it is equal to t1. At this point, the resource referred to by the expressions t1, t2.a, and out.b.a is released, and is therefore no longer tracked.

To be effective, the algorithm needs to be able to prove that **this**.a in CFG (e) is equal to t1 assigned in CFG (a). This requires precise inter-procedural *must-alias* reasoning. TRACKER performs efficient must-alias reasoning using selective equality predicates, *without* relying on whole-program alias analysis as was done in prior work (*e.g.*, [9]) on typestate analysis and resource leakage analysis.[1]

[1]In modern OO languages, construction of a call graph itself is a whole-program analysis; we describe our computation of call graph briefly in Sec. 4.

Consider now the exceptional path 7-6-8-9-11-14-15-exit. On line 15, the expression out refers to the object allocated on line 9, and it is *not equal* to t1, the resource allocated on line 7. At the exit of the procedure, the expression t1 still points to the unreleased resource from line 7, and no other expressions may point to that resource. Since t1 goes out of scope at the procedure exit, we conclude that the resource allocated on line 7 is unreachable, unreleased, and leaks.

What about the path 7-6-5-11-14-exit, where the branch on line 14 is not taken? If the analysis can prove that the false outcome is not possible, the path is infeasible; otherwise, the analysis must report a leak. TRACKER avoids this false positive. What about the path 7-6-14-exit, in which an OutOfMemoryError occurs on line 6? The resource can leak along that path. However, at the user's discretion, TRACKER can suppress problems along exceptional flows associated with fatal exceptions.

2.2 Intra-procedural analysis

We describe our analysis on a control-flow graph containing the following kinds of abstract statements: entry, exit, p = acquire R, release R q, branch c L, p = **new** T, p = q.f,

```
IntraproceduralResourceAnalysis($CFG$, $Specification$)
1  type $Fact$: $SSAVariable \times ResourceType \times State$
2  var $D$: $Statement \rightarrow 2^{Fact}$
3  for $s \in Statement$ do
4      $D(s) \leftarrow \emptyset$
5  for $s \in Statement$ do
6      if $s$ is p = acquire R
7          $a$ = initialState($p$)
8          $\forall t \in succ(s)$: $D(t) \leftarrow D(t) \cup \{\langle p, R, a \rangle\}$
9  while changes in $D$ do
10     $s \leftarrow$ pick from $Statement$
11     $\langle p, R, a \rangle \leftarrow$ pick from $D(s)$
12     case $s$ is release R q
13         if $\neg$isMustAlias($p$, $q$, $a$)
14             $\forall t \in succ(s) : D(t) \leftarrow D(t) \cup \{\langle p, R, a \rangle\}$
15     case $s$ is branch $c$ $L$
16         if isConsistent($c$, $a$)
17             $u = trueSucc(s) : D(u) \leftarrow D(u) \cup \{\langle p, R, a \rangle\}$
18         if isConsistent($\neg c$, $a$)
19             $u = falseSucc(s) : D(u) \leftarrow D(u) \cup \{\langle p, R, a \rangle\}$
20     case $s$ is exit
21         if isUnreachable($a$, $p$)
22             report $p$ as leaking
23     other
24         $a' \leftarrow \ldots$
25         $\forall t \in succ(s) : D(t) \leftarrow D(t) \cup \{\langle p, R, a' \rangle\}$
```

Figure 3: Intra-procedural leak algorithm.

p.f = q, p = q, and invoke. For clarity, we assume that we have rewritten method calls that allocate and release resources abstractly as acquire and release statements. For example, p = **new** FileOutputStream(file) is represented as:

```
p = acquire FileOutputStream
invoke p.<init>(file)
```

The statement p.close() is represented as:

```
release FileOutputStream p
invoke p.close()
```

We assume that each local variable has a single static assignment (SSA) [8]. The abstract statement p = q is included to describe the treatment of ϕ-nodes introduced in SSA conversion, as well as to model transmission of values from actuals to formals in a procedure call. The branch statement has a conditional expression c and a jump target L.

Generic Data-flow Analysis.

Figure 3 shows a generic resource tracking analysis. The algorithm performs iterative data-flow over a powerset lattice of facts of the kind *Fact*, as defined on line 1. A *Fact* is a 3-tuple that consists of an *SSAVariable*, which is an SSA value number; a *ResourceType*, which is the kind of the tracked resource; and a *State*, which represents a finite amount of auxiliary information used to resolve queries the algorithm will make. A concrete description of *State* will be given shortly, but for now assume that it contains predicates over local variables. The algorithm makes use of the following auxiliary functions:

1. initialState($p : SSAVariable$) creates an element of *State* based on the given SSA variable.

2. isMustAlias($p : SSAVariable$, $q : SSAVariable$, $a : State$) evaluates to true if, given the information in a, p equals q. If this check fails, the analysis cannot assume that the variable on which *release* is called strongly [5] closes a resource referred by p. A conservative answer to this query is false.

3. isConsistent($condition : Expression, a : State$) is true if *condition*, which is a conditional expression (*Expression*), does not contradict the information in a. This function is used to prune infeasible paths. A conservative answer to this query is true.

4. isUnreachable($a : State, p : SSAVariable$) returns true if the resource referenced by p may no longer be accessible by any live name, as per information in a. At the exit of a procedure, local variables are assumed to become dead, unless otherwise preserved in a. A conservative answer to this query is true.

Lines 5-8 seed the analysis with initial facts that correspond to resource allocation. The algorithm then propagates these facts through program statements, creating new facts along the way. Lines 24-25 show the effects of pointer statements, such as p.f = q, on *State*; we describe these effects shortly. The algorithm converges because no fact is deleted from the map D, and because each component of *Fact* is finite.

Defining State.

Tracker implements *State* as a set of *must-access-paths* to a tracked resource [13, 15]. A must-access-path is an expression comprised of a variable followed by a (possibly empty) sequence of field names, such that the value of the expression refers to the resource. For example, out.b.a is a must-access-path. Figure 4 shows how a set of must-access-paths, named *in* is transformed by individual program statements. As is customary in flow analysis, *gen* refers to the new must-access-paths added to the set, and *kill* refers to the must-access-paths removed from the set. (We remind the reader that *State*, here represented by set of must-access-paths, is only one component of the data flow domain *Fact*.) In the table of transformations, alias(α,p) checks if p may be an alias of expression α, based on an inexpensive type match.

Because *State* is finite, each time the transformation of must-access-paths set is computed, we must limit the size of the resulting set using a function *filter*. It is necessary to do so for two reasons: (a) in the presence of loops (or recursion), it is possible for access paths to grow indefinitely, and (b) even loop free code might inflate the sets to needlessly large sizes, compromising efficiency. The function filter(a : *State*) empties the set if either any access path in it is longer than a preset limit ("depth"), or the number of access paths in it has exceeded a preset limit ("breadth"). We did not find it useful to trim the set to size instead of emptying it.

Given this implementation of *State*, the auxiliary functions mentioned above are defined as follows. initialState(p) produces a singleton set $\{p\}$. isMustAlias(p, q, a) checks if q is in the must-access-path set a. isConsistent($condition, a$) can be resolved for (only) certain kinds of queries. If a contains $v.\pi$, where π is a possibly empty list of fields, then we know that v cannot be null. Therefore, a *condition* is of the form $v = null$ or $v \neq null$ can be decided exactly. If a contains both v and w, then we can infer $v = w$. Finally, isUnreachable(a, p) is true if a is empty.

Example 1.

Consider the code fragment shown below. We show the facts accumulated by our analysis after each statement to the right.

statement	out = filter$((in - kill) \cup gen)$
p = q.f	$gen = \{p.\pi \mid q.f.\pi \in in\}$ $kill$ = startsWith(p, in)
p.f = q	$gen = \{p.f.\pi \mid q.\pi \in in\}$ $kill$ = startsWith$(p.f, in)$ $\cup$ aliasMatches(p, f, in)
p = **new** T	$kill$ = startsWith(p, in)
p = acquire R	$kill$ = startsWith(p, in)
p = q	$gen = \{p.\pi \mid q.\pi \in in\}$ $kill$ = startsWith$(p.f, in)$
startsWith(p, in)	$\{p.\pi \mid p.\pi \in in\}$
aliasMatches(p, f, in)	$\{\alpha.f \mid \alpha.f \in in \land$ alias$(\alpha, p)\}$

Figure 4: Flow functions for access path sets. π is a possible empty sequence of field names.

p = acquire R	$\langle p, R, \{p\}\rangle$
q.f = p	$\langle p, R, \{p, q.f\}\rangle$
r = q.f	$\langle p, R, \{r, p, q.f\}\rangle$
branch (r == **null**) L1	T: *none*, F: $\langle p, R, \{r, p, q.f\}\rangle$
release r	*none*
L1:	*none*

At the branch statement, isConsistent check tells us that only the fall-through successor is feasible: r, being a must-alias to a resource, cannot be a null pointer. At the release statement, the call to isMustAlias$(p,\ r,\ \{r, p, q.f\})$ succeeds. Consequently, no fact makes it to L1.

Had we used an imprecise and conservative isConsistent, we would have obtained the fact $\langle p, R, \{r, p, q.f\}\rangle$ after L1. Local variables would then be dropped from the state, giving the fact $\langle p, R, \{\}\rangle$ at the exit. This fact would satisfy the query isUnreachable$(\{\}, p)$, resulting in a false positive.

If we had filtered the *State* on line 3 for a breadth limit of 2, we would have conservatively decided that the release on r may not release the resource acquired on line 1: that is, isMustAlias$(r,\ p,\ \{\})$ would be false. Again, this would have resulted in a false positive.

Example 2. Consider the leaky code fragment shown below. It allocates a resource in a loop, but frees only the last allocated instance. The branch after L1 has a non-determnistic condition for which the analysis must answer isConsistent true for both outcomes.

p_1 = **null**	
L1	$\langle p_3, R, \{p_3\}\rangle$, $\langle p_3, R, \{p_2\}\rangle$
$p_2 = \phi(p_1, p_3)$	$\langle p_3, R, \{p_2, p_3\}\rangle$, $\langle p_3, R, \{\}\rangle$
branch $*$ L2	
p_3 = acquire R	$\langle p_3, R, \{p_3\}\rangle$, $\langle p_3, R, \{p_2\}\rangle$
branch **true** L1	
L2	$\langle p_3, R, \{p_2, p_3\}\rangle$, $\langle p_3, R, \{\}\rangle$
release p_2	$\langle p_3, R, \{\}\rangle$

This fragment also illustrates the treatment of ϕ nodes. Consider the path taken through the loop two times and then exiting to L2. The initialization generates $\langle p_3, R, \{p_3\}\rangle$ after the acquire. The generated fact flows to L1, where the ϕ generates $\langle p_3, R, \{p_2, p_3\}\rangle$, using the effect of $p_2 = p_3$. This, in turn, flows out to L2, where it is removed by release.

Next time in the loop body, the acquire statement kills the occurrence of p_3 in $\{p_2, p_3\}$, generating the fact $\langle p_3, R, \{p_2\}\rangle$. Going around the back edge again, this last fact is transformed by the ϕ statement to $\langle p_3, R, \{\}\rangle$. When the transformed fact flows out to L2, it cannot be removed by release. The test isUnreachable$(\{\}, p_3)$ passes, and we report a leak.

Alternative definitions of state. It is worth reiterating the role of *State*: it keeps track of interesting information related to a specific resource allocation. We have presented a specific embodiment of *State*. One may decide to implement the analysis described here using a richer *State* at additional run-time cost: for example, it could track even more predicates to enable more precision in call resolution or in branching. We have not found a compelling need to enrich *State*. Going in the other direction, since there are conservative ways of answering the queries we make on *State*, it is also possible to have a trivial embodiment of it, at the expense of precision. A third alternative is to decouple the *State* from a specific *Fact*, and answer the auxiliary predicates based on globally computed information. For example, one might compute a precise flow- and context-sensitive points-to analysis [25] to answer the queries. We decided against this approach since it has proven difficult to scale such analyses to the sizes of applications we wish to handle.

2.3 Inter-procedural analysis

The algorithm presented in Figure 3 generalizes easily to the inter-procedural case. As is fairly standard, we assume there is an inter-procedural *call* edge from a call statement to each of the possible callees (a static call graph can indicate multiple possible callees), and there is a *return* edge from the exit of a callee to each of the call statements that could have invoked it. For an example, see Fig. 2 where we show call and return edges using dotted lines. A call edge is accompanied by the assignment of formal parameters to values of the corresponding actuals, and likewise, a return edge is accompanied by the assignment of the return value (if any) to the left-hand-side (if any) of the call statement. We implement inter-procedural analysis over this structure using the *IFDS* framework [21].

Relevant Callees. Our algorithm implements the following performance optimization when it encounters a call statement. For a fact $\langle p, R, a\rangle$ that reaches the call instruction, it determines whether the callee method is *relevant*. A method is relevant to a fact if it (or any of its transitive callees) contains a statement that may alter the fact's *State*. The algorithm uses an inexpensive and conservative side-effect computation to determine relevance. If a callee is relevant to a fact, the fact is propagated into the callee. Here, formal parameters of the callee are assigned corresponding values from the actuals using the transformation for a copy statement (p=q). If the callee is not relevant, the fact is propagated only to the successors of the call statement, bypassing the call. This optimization is fruitful, because in practice a lot of methods are auxiliary in nature, and it is wasteful to drag facts along the CFGs of those methods. Any exceptions that can be thrown inside the bypassed callees, and that are not caught locally, are reflected in our IR as exceptional flow edges emanating from the call instruction in the caller's CFG. Thus soundness is not sacrificed when bypassing callees.

Namespace Management. Because SSA variables, as used in Sec. 2.2, are only unique within a single procedure, we need to protect caller value numbers from getting mixed up with callee value numbers. Details of this are routine and are omitted.

2.4 Soundness

Given a sound call graph, TRACKER is sound in that it does not miss true positives. We note, however, that computing a sound call graph for partial programs, or in the presence of reflection, is non-trivial. We describe our *best-effort* call graph construction briefly in Sec. 4.

We also note that we are analyzing open programs: *e.g.*, a library without client code. In such programs, a resource occasionally escapes to the unknown caller either as a return value, or as a field of some longer lived object. We do not call such situations leaks, assuming optimisitically that the (missing) client would release the escaped resources. We could easily implement a pessimistic version of the algorithm, but we find the optimisitic assumption more useful. Our claim to soundness is modulo this assumption.

3. COMPUTING ACTIONABLE REPORTS

This work focuses on producing error reports that are actionable, rather than merely comprehensive. We discuss three ways in which we enhance actionability. First, we prioritize reports based on the structure of access paths sets, so that likely true positives are ranked higher. Second, we cluster reports that arise from leakage of the same underlying resource. Third, we suppress reports that arise from exception flows that are deemed unlikely or uninteresting to the programmer.

3.1 Prioritization

As discussed in Sec. 2.2, our analysis works by tracking sets of must-access paths to unreleased resources and reporting a leak if a tracked set becomes empty. An access path set may become empty in one of two ways: (1) every handle in the set is rooted at a local variable that is going out of scope, or (2) the size of the set, or the length of one the handles, is about to exceed a preset limit. In the first case, the generated leak report is based on a leakage witness—a path through the program's supergraph along which the given resource is not released. These are called *witnessed reports*. In the second case, the analysis is unable to find a witness with the given limits on fact size, so the resulting leak report is called an *assumed report*.

Witnessed and assumed reports obtained by progressively relaxing the limits on the size of tracked facts have two key properties, which our implementation exploits to generate low cost, high quality results. First, for a given limit d on the length of tracked paths and a limit b on the size of tracked sets, witnessed reports $\mathcal{W}_{\langle d,b\rangle}$ are significantly more likely to be true positives than the assumed reports $\mathcal{A}_{\langle d,b\rangle}$ (Sec. 5.1). We therefore prioritize TRACKER's output so that the reports in $\mathcal{W}_{\langle d,b\rangle}$ are ranked higher than those in $\mathcal{A}_{\langle d,b\rangle}$. Second, for a pair of limits $\langle d,b\rangle$ and $\langle d',b'\rangle$, where $d \le d'$ and $b \le b'$, $\mathcal{W}_{\langle d,b\rangle} \subseteq \mathcal{W}_{\langle d',b'\rangle}$ and $\mathcal{A}_{\langle d,b\rangle} \cup \mathcal{W}_{\langle d,b\rangle} \supseteq \mathcal{A}_{\langle d',b'\rangle} \cup \mathcal{W}_{\langle d',b'\rangle}$. That is, the analysis generates more witnessed reports and rules out more false positives when tracking richer facts, which is a straightforward consequence of our definition of a leak. An empirical corollary to this is that the quality of the reports plateaus quickly, so that $\mathcal{W}_{\langle d,b\rangle}$ captures the majority of true positives for a small d and b. As a result, the default configuration for our tool is $d = 3$ and $b = 5$, which

```
1  public Integer getFirst(File in) throws IOException {
2    FileInputStream fis = new FileInputStream(in);
3    Integer ret = null;
4    try {
5      int val = fis.read();
6      ret = new Integer(val);
7    } catch (IOException e) {}
8    fis.close();
9    return ret;
10 }
```

Figure 5: Ignoring fatal exceptions

yields nearly as high a proportion of witnessed (and, hence, highly ranked) true positives as would a slower configuration that uses larger limits.

3.2 Exception Flow

Many resource leaks found in real programs occur along exceptional paths. The program in Fig. 1, for example, may leak the file output stream if the constructor of the stream writer throws an UnsupportedEncodingException. But not all exceptional paths are interesting or likely to be exercised, and a leakage analysis that processes all exceptional edges would generate a large number of reports with little practical value (Sec. 5.2).

To illustrate, consider the program in Fig. 5. The program is, for all practical purposes, free of leaks. It closes the allocated file input stream along all normal paths, and any IOException that may be thrown by the call to read() is caught and ignored. Nonetheless, an analysis that processes *all* exceptional edges would generate a leak report for this program because fis.close() would not be executed if the allocation of the Integer object on line 6 failed due to an OutOfMemoryError. Such a report is not very useful, however. Most programs are not expected to recover from an OutOfMemoryError, and once the program fails, all system resources are automatically released.

Our analysis processes exceptional edges selectively. In particular, we only process the edges associated with *relevant* exception types. The definition of what is relevant is customizable, but is guided by a belief-based heuristic [14] that considers an exception irrelevant unless there is evidence to the contrary. In our implementation, relevant exception types for a method m include all types E such that: (1) m explicitly throws an instance of E via a **throw** statement; (2) m explicitly catches an instance of E in a **catch** block; or (3) E appears in the **throws** clause of m's signature. The set of relevant exception types for a program is simply the union of the relevant exception types for each of the methods in its call graph.

3.3 Nested Resources

It is commonplace in Java to have classes that encapsulate or *wrap* a resource in one of their fields. In PrintWriter and OutputStreamWriter, both of which we used in Fig. 1, the contract says that the output stream passed as an argument to the constructor is closed when close() is called on the encapsulating instance. The following code snippet is therefore free of leaks even though fos.close() is not called directly:

```
FileOutputStream fos = new FileOutputStream(file);
OutputStreamWriter osw = new OutputStreamWriter(fos);
...
osw.close();
```

Unlike leaks of system resources, wrapper leaks are often benign. If a wrapper is encapsulating a memory resource, such as as a ByteArrayOutputStream, then a failure to close the wrapper is uninteresting, as it will be handled by the garbage collector. The same is true if a wrapper encapsulates a system resource that is released independently, as shown below:

```
FileOutputStream fos = new FileOutputStream(file);
OutputStreamWriter osw = new OutputStreamWriter(fos, enc);
...
fos.close();
```

But not all wrapper leaks are uninteresting. If a wrapped system resource is leaking, and the handle to the resource is inaccessible (as in Fig. 1), then only way to actually fix the underlying leak is by releasing the wrapper. We therefore report leaks on wrappers which encapsulate leaking system resources. Since these reports are not independent, we *cluster* them so that all wrapper leaks pertaining to the same underlying resource are in the same cluster. For example, the leaks on the FileOutputStream, OutputStreamWriter and PrintWriter resources in Fig. 1 form a single report cluster.

We cluster reports by post-processing the output of the standard inter-procedural algorithm. Initially, wrapper types seed the analysis just like real resources. Once the analysis terminates, potentially leaking instances are organized into a forest of directed acyclic graphs. The roots of these graphs represent real resources, and edges lead from wrapped to wrapper instances. An edge is created between instances p and q if $q.f$ is a must alias of p, and f is a field that holds a wrapped object. The determination of which fields hold nested resources is a matter or specification. Each disconnected portion of this forerst is presented as a cluster.

4. IMPLEMENTATION

We implemented the techniques presented here in a tool called TRACKER, which is based on the WALA [1] program analysis framework. TRACKER analysis is performed in three stages. First, the tool computes the call graph of the program to be analyzed. Because many of the standard call graph construction algorithms (*e.g.*, 0-CFA) neither scale to large applications nor work well for partial programs, we base our call graphs directly on the program's class hierarchy, using type information to determine targets of virtual dispatches. For efficiency, the computed call graph includes only a necessary subset of the program's methods. In particular, a given method m is included only if it contains a resource allocation instruction or it transitively calls a method that contains such an instruction. Each included method must also be reachable from a public entry point (*i.e.*, a public method of a public class). After it has constructed the call graph, TRACKER performs the core resource tracking analysis (Sec. 2), which generates a set of witnessed and assumed leak reports (Sec. 3.1). The final stage of the analysis involves clustering and filtering (Sec. 3.3) of these reports for improved actionability.

5. EMPIRICAL EVALUATION

To evaluate TRACKER, we conducted a series of experiments on the five open source programs listed in Table 1. These include a build tool (ANT 1.7), an image manipulation toolkit (BATIK 1.6), a reporting system for web applications (BIRT 2.5), a peer-to-peer bit torrent client (FROSTWIRE 4.17), and a web server (TOMCAT 6.0). The table shows the size of each benchmark, given as the number of classes and methods. Also shown are the number of resource allocation sites in each program and the size of its call graph. The latter is measured by the number of included methods, the number of classes containing those methods, and the time taken to compute the graph.

We conducted four experiments on the candidate programs. The first three experiments were designed to evaluate the effectiveness of our ranking heuristic, exception-filtering mechanism and clustering techniques. The fourth compares TRACKER to FINDBUGS [4]. Unless stated otherwise, TRACKER was configured to filter exceptions, cluster all reports, filter non-actionable wrapper reports, and to track at most 10 access path of length 10 or less per resource. Tracking facts larger than $\langle 10, 10\rangle$ did not change the output of the tool for any of the subject programs. We refer to this configuration of the tool as the *baseline configuration* or BASE. Report counts shown in the tables and graphs refer to the number of report clusters, unless otherwise specified. The same set of resource acquire-release specifications was used for each experiment: the stream resources in the java.io package and the database resources in the java.sql package. All experiments were performed on 2.4 GHz Intel Core Duo machine with 4 GB of memory.

	Program Size		# Resources		Call Graph Size		
	classes	methods	system	wrapper	classes	methods	sec
Ant 1.7	6,952	63,033	148	242	1,440	7,990	18
Batik 1.6	11,187	97,048	47	97	2,381	10,365	24
Birt 2.5	69,293	572,744	200	305	9,900	53,802	304
Frostwire 4.17	16,279	127,363	120	204	3,676	19,856	64
Tomcat 6.0	9,981	102,220	124	167	2,273	13,487	30

Table 1: Benchmark statistics.

5.1 Report Prioritization and Actionability

Goals and methods. To assess the effectiveness of our ranking heuristic, we configured the base TRACKER with five different limits on the size of tracked facts and used it to compute five corresponding sets of reports for each benchmark. The five limits include the base limit $\langle 10, 10\rangle$ and four stricter limits: $\langle 1, 3\rangle$, $\langle 3, 3\rangle$, $\langle 3, 5\rangle$, and $\langle 5, 5\rangle$. We manually classified the base report sets into *true positives* (*TP*) and *false positives* (*FP*). This classification was then used to automatically classify the remaining report sets, taking advantage of the fact that TRACKER's output is sound (so it always includes all true positives) and monotonic (so it includes fewer false positives as limits on fact size are increased).

Results. The results are presented in Fig. 6. We show one chart per benchmark, each with one bar per report set. The height of a bar indicates the total number of reports in the given report set. The height of the shaded regions within a bar indicates the number of witnessed true positives (black), assumed true positives (white pattern on black background), witnessed false positives (white), and assumed false positives (black pattern on white background). For example, the $\langle 3,5\rangle$ report set for ANT includes a total of 95 reports, which consist of 54 true positives (47 witnessed and 7 assumed) and 41 false positives (9 witnessed and 32 assumed). The graph on the bottom right shows the time, in seconds, taken to compute each report set after the call graph has been constructed.

Discussion. The data shown in Fig. 6 reveals three key trends. First, due to the soundness and monotonicity of TRACKER, the total number of true positives for each benchmark remains constant and the number of false positives decreases monotonically as limits on fact size are relaxed. In practice, this means that the user of the tool can easily control the trade-off between resource consumption and report quality. Second, for most limits on fact size, true positives comprise a much higher proportion of witnessed reports than of assumed reports: $\frac{|\mathcal{W}_{\langle d,b\rangle} \cap TP|}{\mathcal{W}_{\langle d,b\rangle}} \gg \frac{|\mathcal{A}_{\langle d,b\rangle} \cap TP|}{\mathcal{A}_{\langle d,b\rangle}}$. In other words, witnessed reports, which are ranked as "high priority" by TRACKER, have a higher probability of being true positives than the lower ranked assumed reports. Third, the number of witnessed true positives obtained with the default limit of $\langle 3,5\rangle$ on fact size captures the majority of the witnessed true positives obtained with more generous limits. The tool's default configuration therefore offers a good trade-off between performance and quality for a wide range of applications.

5.2 Exception Flow Filtering

Goals and methods. To evaluate the effect of exception filtering on report quality, we applied two configurations of TRACKER to all five benchmarks: the baseline configuration that filters exceptional edges as described in Sec. 3.2 and a variant configuration that processes all edges indiscriminately. Reports generated by the variant but not the baseline were manually classified into true and false positives.

Results. Table 2 summarizes our findings. For each benchmark, we show the total number of new true and false positives, as well as the fraction of the new results with respect to the baseline output. For example, applying the variant configuration of TRACKER to ANT yielded 44 new reports, only two of which were true positives. This comprises four percent of the true positives found by the baseline configuration.

Discussion. Overall, we found that exception filtering significantly improves the quality of TRACKER's results. Disabling the filter introduced a large percentage of false positives and only a negligible fraction of true positives in all but one benchmark, where it had no effect. Most of the additional false positives were due to low-likelihood exceptional edges within **try**–**catch**–**finally** blocks. The additional true positives were all due to exceptions that our filter deems irrelevant but that we thought likely to occur in practice.

5.3 Report Clustering and Wrappers

Goals and methods. To measure the effects of report clustering and of distinguishing between system resources and wrappers, we compared the output of our baseline configuration to that of the ALL-REAL configuration. The latter, unlike BASE, treats wrappers as real system resources. The comparison of the two configurations involved simply counting the number of report clusters and the number of individual reports in each result set. We did not classify the leaks reported by ALL-REAL but not by BASE, since we consider all reports on wrappers that do not (transitively) wrap a real resource to be non-actionable.

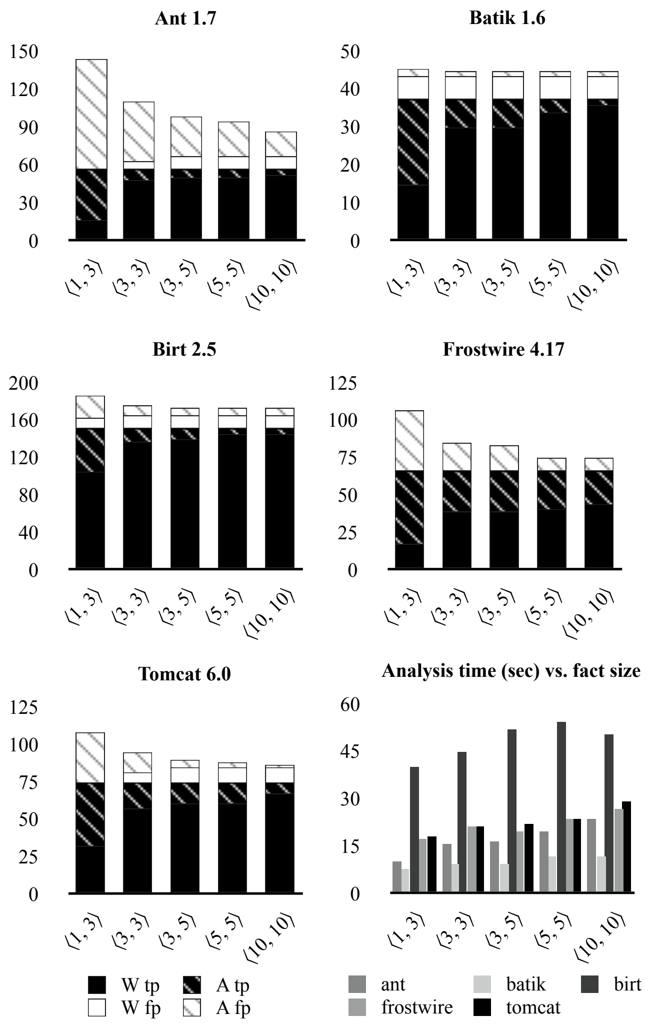

Figure 6: Performance and quality of reports for a sampling of fact sizes.

Results. Figure 7 presents the results. The chart shows the number of clusters and the number of individual reports, both BASE and ALL-REAL, for each benchmark. For example, BASE generated a total of 151 individual reports for ANT, which were grouped in 83 clusters. The ALL-REAL configuration, on the other hand, produced 238 individual reports and 148 clusters.

Discussion. Comparing the output of BASE and ALL-REAL, we find that, on average, ALL-REAL generates twice as many reports and clusters as BASE. None of the additional reports are, in our view, actionable. We therefore believe that a practical resource leakage analysis must distinguish between system resources and wrappers. For both BASE and ALL-REAL, clustering decreases the number of distinct leaks that the user needs to examine by 15 to 45 percent. In our experience, clustering was particularly helpful for triaging complex inter-procedural leaks that involve a single system resource and several different wrappers.

	TP	FP	TP / base TP	FP / base FP
Ant 1.7	2	42	0.04	1.45
Batik 1.6	0	0	0.00	0.00
Birt 2.5	1	11	0.01	0.50
Frostwire 4.17	2	23	0.03	2.56
Tomcat 6.0	1	19	0.01	1.58

Table 2: Effects of exception flow filtering.

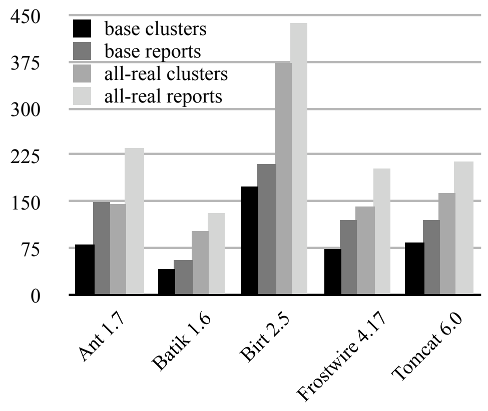

Figure 7: Effects of report clustering on the output of the base and all-real configurations of the analysis.

5.4 Comparison with Related Techniques

Goals, methods and results. To evaluate TRACKER against an existing, widely-used tool, we applied FINDBUGS to our five subject programs, classified the output into true and false positives, and then compared the resulting classified reports to those of TRACKER BASE. The results are presented in Table 3. For each benchmark and each class of leaks, we show the number of reports that were found by both tools, as well as the reports found by only one of the tools. For example, in `ant`, TRACKER found 46 true positives not found by FINDBUGS, while all 8 true positives reported by FINDBUGS were also reported by TRACKER.

Discussion. Overall, TRACKER found significantly more true positives than FINDBUGS. We believe this is because FINDBUGS relies on heuristics to report only the most compelling defects that it could reason about intra-procedurally. FINDBUGS reported fewer false positives, but the great majority of TRACKER's false positives were the low-ranked assumed reports. The few true positives that were detected by FINDBUGS but not by our tool were all due to engineering choices rather than fundamental limitations. For example, the FROSTWIRE true positives missed by TRACKER are located in classes that were not analyzed because of the way we determine the scope of a program's class hierarchy.

6. RELATED WORK

Our work has targeted problems relating to leaks of system resources such as sockets and file descriptors. Much related work targets memory leaks, either with explicit memory management (*e.g.*, `malloc/free`) or garbage collection. Whether considering memory leaks or system resource leaks, a static analysis must reason about a liveness property for

	True Positives			False Positives		
	FB	TR	BOTH	FB	TR	BOTH
Ant 1.7	0	46	8	4	29	0
Batik 1.6	1	26	11	18	5	2
Birt 2.5	8	97	53	8	16	6
Frostwire 4.17	3	61	5	1	9	0
Tomcat 6.0	2	63	11	4	12	0

Table 3: Comparison of tracker and findbugs.

objects in certain states, in the presence of potentially complex aliasing.

Much previous work has considered type systems for region-based memory management (*e.g.*, [23]), whereby a type system prevents memory leaks and enables an implementation of efficient storage reclamation. Alternatively, approaches based on escape analysis (*e.g.*, [25]) typically employ sophisticated, modular alias analysis to identify potential sources of leaks. Other relevant tools with analysis for object liveness and memory leaks include Free-Me [16], Clouseau [17], and Uno [20].

Qualitatively, bug-finding for system resource leaks has a different flavor from addressing general memory leaks. First, system resource management usually involves a small subset of a large program, whereas nearly every operation in an object-oriented language involves dynamically allocated memory. Second, in a bug-finding tool, it is reasonable to suppress findings to avoid false positives, whereas many applications in system memory management need conservative analysis.

6.1 Analysis Tools

Weimer and Necula [24] studied system resource leak problems in Java programs due to incorrect exception handling. This work presents a simple path-sensitive intraprocedural static analysis to find resource leak bugs. Weimer and Necula's analysis is unsound; nevertheless, they report that the analysis still finds hundreds of problems with open-source Java programs. In contrast to our work, Weimer and Necula's analysis does not consider aliasing, inter-procedural paths, or nested resources.

Static analysis of resource leaks relies on analysis of a *dynamic location liveness* property as defined by Shaham *et al.* [22]. Shaham *et al.* presented a conservative static analysis based on canonical abstraction to verify safety of synthesized `free` operations for dynamically allocated objects. A similar analysis could be used to insert `dispose` operations to prevent resource leaks. Charem and Rugina [7] describe a similar approach with a less expensive analysis. The CLOSER [12] performs a modular, flow-sensitive analysis to determine "live" system resources at each program point. In contrast to these approaches, TRACKER neither proposes nor checks safety of remediation at particular program points, and it relies on significantly less expensive alias abstractions.

Our analysis tracks a heap abstraction based on k-limited access-paths using flow-sensitive, context-sensitive interprocedural analysis. Access path alias analysis has been used in many previous works, including [19, 11, 5, 15].

Tools based dynamic instrumentation can detect resource leaks, by checking that the appropriate disposal methods are called when an object is reclaimed. The QVM [3] takes this approach, engineering a virtual machine where the garbage collector cooperates to allow dynamic instrumentation to detect object reclamation events. On the other hand, approaches based on bytecode instrumentation or aspects (*e.g.*, [2, 6]) better suit safety properties than these liveness properties, without cooperation from the garbage collector to detect object death.

6.2 Languages and Type Systems

Java's finalizers provide a language construct designed to help the programmer free resources. Section 6 of [24] reviews problems with destructors and finalizers in practice. Java's finalizers are particularly problematic for dealing with finite resources, since the garbage collector provides no guarantees for timely finalizer execution.

Many languages provide constructs to associate a resource with a lexical scope, and ensure that the resource is closed when the scope finishes. For example, C# provides a `using` statement to guarantee timely disposal of resources at the end of a lexical scope [18]. In our experience, many of the resource leaks TRACKER found in Java code could be eliminated if a similar lexical construct were available. However, these solutions do not apply to code that caches resource pointers in a heap, untied to a convenient lexical scope.

Weimer and Necula [24] propose a language construct for Java called *compensation stacks*. A compensation stack associates each logical resource allocation with a scope, and guarantees that a clean-up action (a closure) executes when the program exits the scope. The scope can be tied to a lexical boundary such as a method, or manipulated as a first-class object, to support more general control structures.

The Vault programming language [10] provides a type system which can rule out certain types of resource leaks. The Vault type system allows the programmer to specify function postconditions that ensure functions cannot allocate and leak resources.

7. CONCLUSION AND FUTURE WORK

We presented a static analysis tool that reports resource leakage defects in Java applications. There are many engineering challenges in building such a tool, including the generic problems of scalability and precision, as well as the specific problems of dealing with exceptions and wrappers. Our contribution is in overcoming these challenges, using a blend of existing and new techniques. An empirical evaluation of the tool showed its overall effectiveness, as well as the importance of the techniques we discussed in the paper.

Several directions for future work have presented themselves. One is the problem of discovering specifications automatically, especially with regards to identifying which pairs of types are related by a wrapper-wrappee relationship. Another promising direction is to extend the leak detection algorithm to automatically suggest code refactorings.

Acknowledgement. We are indebted to Stephen Fink for providing valuable comments on the paper as well as for helping us with the FINDBUGS experiments.

8. REFERENCES

[1] T. J. Watson Libraries for Analysis. `http://wala.sourceforge.net/`.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05*, 2005.

[3] M. Arnold, M. Vechev, and E. Yahav. QVM: an efficient runtime for detecting defects in deployed systems. In *OOPSLA '08*, pages 143–162, 2008.

[4] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[5] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI '90*, pages 296–310, 1990.

[6] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *OOPSLA '07*, 2007.

[7] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *ISMM '04*, pages 85–96, 2004.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[9] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

[10] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01*, pages 59–69, 2001.

[11] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94*, pages 230–241, 1994.

[12] I. Dillig, T. Dillig, E. Yahav, and S. Chandra. The CLOSER: automating resource management in Java. In *ISMM '08*, pages 1–10, 2008.

[13] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA '04*, pages 12–22, 2004.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.

[15] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144, 2006.

[16] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: a static analysis for automatic individual object reclamation. In *PLDI '06*, pages 364–375, 2006.

[17] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI '03*, pages 168–181, 2003.

[18] A. Hejlsberg, P. Golde, and S. Wiltamuth. *C# Language Specification.* Addison Wesley, Oct. 2003.

[19] N. D. Jones and S. S. Muchnick. *Flow analysis and optimization of Lisp-like structures.* Prentice-Hall, 1981.

[20] K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. *SIGPLAN Not.*, 42(10):423–440, 2007.

[21] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[22] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. *Sci. Comput. Program.*, 58(1-2):264–289, 2005.

[23] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value Lambda-calculus using a stack of regions. In *POPL '94*, pages 188–201, 1994.

[24] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. *SIGPLAN Not.*, 39(10):419–431, 2004.

[25] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. *SIGPLAN Not.*, 34(10):187–206, 1999.